Research report 192

LIST PROCESSING IN PARALLEL

TOM AXFORD

AND

MIKE JOY

A new model of list processing is proposed which is well suited to parallel implementation. Its main primitive functions are: *concatenate,* which concatenates two lists; *split,* which partitions a list into two parts; and *length,* which gives the number of elements in a list. In the commonly used CREW P-RAM model of parallel computation, common high-level operations on lists such as *map* and *reduce* take O(log n) time in the new model as against O(n) time in the conventional head-tail-cons model of list processing.

Parallel simulation of a number of simple programs in the functional language FLIC, using the new model of lists, gives parallel execution times consistent with this analysis. The programs tried, use the high-level functions *map, filter* and *reduce*, together with an implementation of Hoare's *quicksort.*

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

August 1991

List Processing in Parallel*

Tom Axford

School of Computer Science,
University of Birmingham,
PO Box 363,
Birmingham,
B15 2TT,
U.K.

E-mail: T.H.Axford@bham.ac.uk

Mike Joy

Department of Computer Science,
University of Warwick,
Coventry,
CV4 7AL,
U.K.

E-mail: msj@dcs.warwick.ac.uk

August 1991

Abstract

A new model of list processing is proposed which is well suited to parallel implementation. Its main primitive functions are: *concatenate*, which concatenates two lists; *split*, which partitions a list into two parts; and *length*, which gives the number of elements in a list. In the commonly used CREW P-RAM model of parallel computation, common high-level operations on lists such as *map* and *reduce* take O(log n) time in the new model as against O(n) time in the conventional head-tail-cons model of list processing.

Parallel simulation of a number of simple programs in the functional language FLIC, using the new model of lists, gives parallel execution times consistent with this analysis. The programs tried use the high-level functions *map*, *filter* and *reduce*, together with an implementation of Hoare's *quicksort*.

1 Introduction

Lists have been used as basic data structures within programming languages since the 1950s. The most elegant and successful formulation was in Lisp [5] with its primitive functions *car*, *cdr* and *cons*, often now

*A shorter version of this paper has been submitted to Information Processing Letters.

referred to by the more meaningful names of *head*, *tail* and *cons* respectively. Lisp and its model of list processing based on the *head*, *tail* and *cons* primitives have given rise to a large number of programming languages over the three and a half decades since Lisp was invented; for example, following closely to the pure Lisp tradition are ML[13], Miranda[12] and Haskell[4].

The success of the Lisp model of list processing is due to a combination of its semantic elegance on the one hand and its simplicity and efficiency of implementation on the other.[1] In the context of functional languages particularly, it has given rise to a style of programming which is clear, concise and powerful. This style is well documented in many publications, for example [3].

Despite the often proclaimed advantages of functional languages for parallel programming [7], there has been very little progress in constructing really worthwhile parallel implementations of them. A large part of the problem lies in the difficulty of obtaining efficient parallel representation of the traditional head-tail-cons model of list processing. Many recent functional languages, such as Miranda and Haskell, have this model intimately woven into the language through features such as pattern matching [11]. In these languages, although it is possible, it is somewhat unusual to write programs that do not depend heavily on pattern matching and hence on the traditional list-processing model. Hence, if this model cannot be implemented efficiently in parallel, most programs are unlikely to be any better.

There are, however, many list operations for which it is easy to envisage a very efficient parallel implementation, particularly the higher-level operations such as *map* and *reduce*, which operate on all elements of the list, not just on a single element. Therefore, the problem is not inherent in the semantics of list processing or the concept of a list itself, but rather with the choice of a set of primitive functions, and the (usually implicit) assumption that the implementation executes these in constant time (i.e. independently of the lengths of the lists involved).

In this paper, we propose a new model of list processing based on a different set of primitive functions, chosen to be efficiently implementable on parallel architectures, but preserving the usual semantics of lists. Programs that use pattern matching on lists or explicitly refer to *head*, *tail* and *cons* will need major rewriting to use the new model. On the other hand, programs that do not use these primitives, but instead use purely high-level library functions may not require any changes at all.

2 The Model

The following six functions are chosen as the primitive functions of the model (the first is actually a constant, or a function which takes no arguments).

(i) `[]` is the empty list.

(ii) `singleton x` (or, alternatively, `[x]`) is the list which contains a single element, `x`.

(iii) `concatenate s t` (or, alternatively, `s++t`) is the list formed by concatenating the lists `s` and `t`.

(iv) `split s` is a pair of lists got by partitioning the list `s` into two parts. It is defined only if `s` contains at least two elements. Both lists are non-empty.

If `s` contains more than two elements, the result of applying `split` is non-deterministic, i.e. there is more than one acceptable solution and an implementation is free to choose which of these to give as the result.

(v) `length s` (or, alternatively, `#s`) is the number of elements in the list `s`.

(vi) `element s` is the only element present in the singleton list `s`. This function is undefined for lists which contain either more or less than one element.

[1] In the early development of Lisp, efficiency of implementation was a major concern, while the desire for an elegant and coherent semantic model of list processing was much less pressing. Nevertheless, the reason that Lisp was more successful than its list-processing competitors almost certainly had a lot to do with McCarthy's perceptive choice of the basic routines that operated on lists. [6]

3 Algebraic Specification

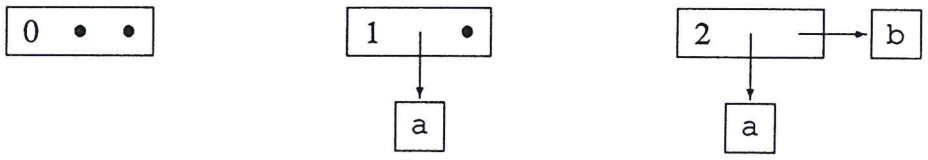
The algebraic properties of the primitive functions that can be used to specify the semantics of the model are as follows:

- `length [] = 0`
- `length [x] = 1`
- `length (s++t) = length s + length t`
- `element [x] = x`
- `s++[] = []++s = s`
- `s++(t++s) = (s++t)++u`
- `split([x]++[y]) = ([x],[y])`
- `length u >= 2, split u = (s,t)` *implies* `s++t = u, length s >= 1, length t >= 1`

4 Representation in the Computer

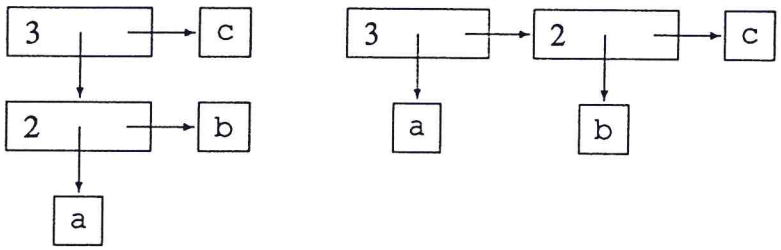
A list is represented as a binary tree. Each node in the tree is either a branch node or a leaf node. Each leaf node contains an element of the list. Each branch node contains two pointers, the left one points to the first part of the list, while the right points to the second part of the list. Ideally these two parts of the list should be approximately equal in length (i.e. the tree should be be balanced), but that is not a requirement for correctness of the representation, it affects only the performance.

The representations of the empty list (`[]`), a singleton list (`[a]`), and a list of two elements (`[a,b]`) are:



The • denotes a nil pointer and occurs only in lists containing no elements, or just a single element.

Two alternative representations of the list `[a,b,c]` are:



The more elements the list contains, the more different tree structures are possible. All are equally valid and will give exactly the same results, although the performance of a program may depend upon how well-balanced the tree is.

With the above representation, it is easy to program implementations of all six primitive functions that will execute in constant time, irrespective of the lengths of the lists involved. None of these primitives offers any scope for parallelism, however. That comes with the implementation of higher-level list-processing functions.

5 High-Level Functions

Most common high-level list-processing functions such as `map` and `reduce` can be easily programmed in terms of the new primitives, using a divide-and-conquer strategy. The structure of divide-and-conquer programs makes them particularly well suited to parallel implementations on a very wide variety of parallel architectures.[1, 2]

Most of the following functions are identical in specification to functions in the standard library of Haskell or Miranda. The parallel performance of each is estimated by analysing the program in the usual CREW P-RAM model (concurrent read, exclusive write, parallel random access memory).

The programs below make use of the function `short` which tests if a list contains exactly one element:

```
>  short s = length s == 1;
```

All program fragments are expressed in a simple functional language pseudocode which is essentially a very small subset of Miranda and Haskell. The usual arithmetic and logic operators are used, except that the relational operator that tests for equality is denoted by == (as in C) and conditional expressions are denoted by `IF..THEN..ELSE..FI`. Each program line begins with the symbol >.

5.1 Map

The function `map` applies a unary function, `f`, to each element of a list, `s`. Thus:

```
map f [x1, ..., xn] = [f x1, ..., f xn]
```

It can be programmed as follows:

```
>  map f s =
>     IF short s THEN [f(element s)]
>     ELSE map f s1 ++ map f s2 FI
>     WHERE (s1,s2) = split s;
```

The execution time for a well-balanced list is O(log n) in parallel, O(n) serially.

5.2 Reduce

The function `reduce` takes two arguments: a binary function `f` and a list `s`. It reduces the list to a single value by applying the function `f` repeatedly between the elements. It is not defined on an empty list.

For example, if `f=(+)` then it adds up all the elements in the list:

```
reduce (+) [x1, ..., xn]  =  x1+ ... +xn
```

The result is non-deterministic unless `f` is associative, but a parallel implementation can follow whatever order of operations gives the best performance. A program for `reduce` is:

```
>  reduce f s =
>     IF short s THEN element s
>     ELSE f(reduce f s1)(reduce f s2) FI
>     WHERE (s1,s2) = split s;
```

The execution time for a well-balanced list is O(log n) in parallel, O(n) serially.

5.3 Filter

This function finds the subset consisting of all elements satisfying a given condition, and returns these elements in the form of a list in the same order as they occurred in the original list. A program for it is:

```
>  filter p s =
>    IF short s
>    THEN IF p(element s)
>         THEN s ELSE [] FI
>    ELSE filter p s1 ++ filter p s2 FI
>    WHERE (s1,s2) = split s;
```

The execution time of `filter` for well-balanced lists is O(log n) in parallel, O(n) serially.

5.4 Quicksort

Hoare's quicksort algorithm can be programmed in the new model as follows (assuming a list of distinct numbers to be sorted into increasing numerical order).

```
>  quicksort s =
>    IF short s THEN s
>    ELSE quicksort s1 ++ quicksort s2 FI
>    WHERE
>    (s1,s2) = reduce f (map g s);
>    g x = IF x <= median
>          THEN ([x],[])
>          ELSE ([],[x]) FI;
>    f (t1,t2)(u1,u2) = (t1++u1,t2++u2);
>    median = (first s + last s)/2;
>    first s = IF short s THEN element s
>              ELSE first s1 FI
>              WHERE (s1,s2) = split s;
>    last s = IF short s THEN element s
>             ELSE last s2 FI
>             WHERE (s1,s2) = split s;
```

The median is estimated in a simple way that is suitable for numerical data. It suffices for purposes of illustration.

The performance of the program on a well-balanced list should be O((log n)^2) in parallel if the estimate of the median is, on average, reasonably good. Under similar circumstances, but with serial execution, the time is O(n log n).

6 Parallel Simulation

In order to show that programs in the new model do in fact have the expected performance, we ran a number of programs on a simulated parallel processor.

6.1 The Graph Reduction Machine

The parallel processor we simulated is a shared-memory multiprocessor performing lazy combinator reduction.

Input to the reducer is FLIC [8], which is first of all translated to an acyclic graph representing an equivalent lambda-expression. This graph also contains primitives (integers, reals and sum-products) and operators (which broadly correspond to the FLIC set of operators), together with the Y combinator to indicate recursion.

An extra operator, `PAR` [9], defined by:

`PAR x y = y,` but with a parallel task spawned to evaluate `x`,

is included to enable parallelism to be specified.

The standard abstraction algorithm [10] is then performed to translate the graph into one containing the Turner combinators (S, K, I, B, C, S′, B′, C′) and no lambdas. Acyclic graph reduction is then used to evaluate the resulting graph.

6.2 Parallel Evaluation Strategy

We assumed that the graph would reside in shared memory, and could be read from or written to by any of the N processors. The evaluation would proceed for a number of cycles; during each cycle each active processor would perform one single reduction step. In the divide-and-conquer paradigm used by the all the programs, each time the problem is split in two, a new process is spawned and sent to a free processor which then proceeds to evaluate the subgraph representing that part of the program.

With each processor are associated two stacks. The first keeps track of the operators performing the graph reductions, together with the graph nodes which are their arguments. The second records the nodes on the 'spines' within the graph, in order that they can be overwritten when required.

When a processor pushes a node onto the second stack, it is tagged as 'locked' by that processor, and cannot be overwritten by another processor until the tag is removed. If a processor attempts to push a node onto the second stack, and that node is already locked, the processor will go into a 'waiting' state until the lock is removed. In certain trivial situations the waiting process will terminate.

Reference count garbage collection is employed. Any attempt to de-reference a locked node results in the locking process being 'killed'.

One processor, which commences the graph reduction, is responsible for all input and output, and cannot be killed. In the absense of any more processors, that processor will fully reduce the graph itself.

6.3 The Test Programs

The four programs used for the performance simulations were the following:

```
map id [1..n]
reduce (+) [1..n]
filter ((==)0) [1..n]
quicksort [n..1]
```

The FLIC code that was used for each of these is shown below.

6.3.1 Map

```
run (map (\x x))
map (\p \s CASE 3
    NIL
    (singleton (p (element s)))
    ((\a1 \a2 PAR a2 (cat a1 a2)) (map p (SEL 3 1 s))
                                  (map p (SEL 3 2 s)))
    s)
length (\x CASE 3 0 1 (SEL 3 0 x) x)
cat (\x \y IF (INT= 0 (TAG x)) y
    (IF (INT= 0 (TAG y)) x
        (PACK 3 2 (INT+ (length x) (length y)) x y)))
element (SEL 1 0)
singleton (PACK 1 1)
```

6.3.2 Filter

```
run (filter (INT= 0))
length (\x CASE 3 0 1 (SEL 3 0 x) x)
cat (\x \y IF (INT= 0 (TAG x)) y
    (IF (INT= 0 (TAG y)) x
        (PACK 3 2 (INT+ (length x) (length y)) x y)))
element (SEL 1 0)
filter (\p \s CASE 3
    NIL
    (IF (p (element s)) s NIL)
    ((\a1 \a2 PAR a2 (cat a1 a2))
        (filter p (SEL 3 1 s)) (filter p (SEL 3 2 s)))
    s)
```

6.3.3 Reduce

```
run (reduce INT+)
reduce (\p \s CASE 3
    ABORT
    (element s)
    ((\a1 \a2 PAR a2 (p a1 a2))
        (reduce p (SEL 3 1 s)) (reduce p (SEL 3 2 s)))
    s)
length (\x CASE 3 0 1 (SEL 3 0 x) x)
cat (\x \y IF (INT= 0 (TAG x)) y
    (IF (INT= 0 (TAG y)) x
        (PACK 3 2 (INT+ (length x) (length y)) x y)))
element (SEL 1 0)
```

6.3.4 Quicksort

```
run qs
reduce (\p \s CASE 3
    ABORT
```

```
    (element s)
    ((\a1 \a2 PAR a2 (p a1 a2))
    (reduce p (SEL 3 1 s)) (reduce p (SEL 3 2 s)))
    s)
element (\x CASE 3 ABORT (SEL 1 0 x) ABORT x)
map (\f \data CASE 3
    NIL
    (singleton (f (element data)))
    ((\a1 \a2 PAR a2 (cat a1 a2))
        (map f (SEL 3 1 data)) (map f (SEL 3 2 data)))
    data)
length (\x CASE 3 0 1 (SEL 3 0 x) x)
cat (\x \y IF (INT= 0 (TAG x)) y (IF (INT= 0 (TAG y)) x
    (PACK 3 2 (INT+ (length x) (length y)) x y)))
singleton (PACK 1 1)
split (TUPLE 2)
qs (\s CASE 3 NIL s
    ( (\pair ((\a1 \a2 PAR a2 (cat a1 a2))
        (qs (SEL-TUPLE 2 0 pair))
            (qs (SEL-TUPLE 2 1 pair))))
        ( reduce f (map (g  (median (first s) (last s)) ) s))) s)
g (\medn \x IF (INT> x medn) (split (singleton x) NIL)
    (split NIL (singleton x)))
f (\sp1 \tp1 split
    (cat (SEL-TUPLE 2 0 sp1) (SEL-TUPLE 2 0 tp1))
    (cat (SEL-TUPLE 2 1 sp1) (SEL-TUPLE 2 1 tp1)))
median (\x1 \xn (INT/ (INT+ x1 xn) 2))
first (\s CASE 3 ABORT (element s) (first (SEL 3 1 s)) s)
last (\s CASE 3 ABORT (element s) (last (SEL 3 2 s)) s)
```

6.4 Simulation Results

In the tables below, the figures are execution times for various values of n (the length of the list). Time is defined here simply as the number of reductions (in the usual lambda-calculus sense), each reduction being assumed to take the same real time in the parallel simulation.

Program	Parallel Time							
	n=2	n=4	n=8	n=16	n=32	n=64	n=128	n=256
`map id [1..n]`	589	720	848	976	1104	1231	1358	1485
`reduce (+) [1..n]`	251	311	371	429	487	545	603	661
`filter ((==)0) [1..n]`	470	568	663	757	851	945	1039	1133
`quicksort [n..1]`	5809	8185	10629	13169	15808	18541	21374	24280

Program	Sequential Time							
	n=2	n=4	n=8	n=16	n=32	n=64	n=128	n=256
`map id [1..n]`	751	1115	1615	2387	3703	6107	10687	19619
`reduce (+) [1..n]`	265	373	543	837	1379	2417	4447	8461
`filter ((==)0) [1..n]`	569	819	1171	1727	2691	4471	7883	14559
`quicksort [n..1]`	7519	12150	18248	27862	45672	82202	160828	332958

These results clearly show that the parallel time varies approximately as $\log n$ for all cases except `quicksort` which varies as $(\log n)^2$, as expected. The maximum number of processors used is n for each of `map`, `reduce` and `filter`, and $2n$ for `quicksort`.

7 Conclusions

The proposed new model of list processing overcomes one of the major obstacles to achieving good parallel implementations of most functional programming languages. Unfortunately, such languages usually have the conventional head-tail-cons model of list processing integrated into the language and a minor revision of such languages would be needed to use the new model. Many programs written in such languages will need fairly major re-design, however. On the other hand, programs that make use of higher-order functions and the standard library of list-processing functions, and that avoid using the head-tail-cons primitives directly, should require little or no change to benefit.

Initial experiments with an implementation of an interpreter for the functional language intermediate code, FLIC, using simulated shared memory parallelism have given performance estimates fully consistent with that expected from our program analysis. Further work is being carried out to determine the parallel performance that can be achieved on a wider variety of applications and in more realistic circumstances.

References

[1] T.H. Axford, An Elementary Language Construct for Parallel Programming, *ACM SIGPLAN Notices* **25**(7) (1990) 72–80.

[2] T.H. Axford, An Abstract Model for Parallel Programming, Research Report CSR-91-5, School of Computer Science, University of Birmingham, 1991.

[3] R. Bird and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, London, 1988).

[4] P. Hudak and P. Wadler (eds.), Report on the Programming Language Haskell, Internal Report, Department of Computer Science, Yale University, 1990.

[5] J. McCarthy, A Micro-Manual for Lisp – Not the Whole Truth, *ACM SIGPLAN Notices* **13**(8) (1978) 215–216.

[6] J. McCarthy, History of Lisp, *ACM SIGPLAN Notices* **13**(8) (1978) 217–223.

[7] S.L. Peyton Jones, Parallel Implementations of Functional Programming Languages, *Computer Journal* **32**(2) (1989) 175–186.

[8] S.L. Peyton Jones and M.S. Joy, FLIC – a Functional Language Intermediate Code, Research Report 148, Department of Computer Science, University of Warwick, Coventry, 1989.

[9] P. Roe, Some Ideas On Parallel Functional Programming, in *Functional Programming, Glasgow 1989* (eds. K. Davis and J. Hughes), (Springer-Verlag, 1990).

[10] D.A. Turner, A New Implementation Technique for Applicative Languages, *Software – Practice and Experience*, **9** (1979) 31–49.

[11] D.A. Turner, Recursion Equations as a Programming Language, in *Functional Programming and its Applications* (eds. J. Darlington, P. Henderson and D.A. Turner), (Cambridge University Press, 1982).

[12] D.A. Turner, Miranda: A Non-Strict Functional Language with Polymorphic Types, in Proceedings of FPCA'89 (ed. J-P. Jouannaud), *Lecture Notes in Computer Science* **201**, 1–16, (Springer-Verlag, Berlin, 1985).

[13] A. Wikstrom, *Functional Programming Using Standard ML*, (Prentice-Hall, London, 1987).